Sissejuhatus infotehnoloogiasse

T.Tammet
TTÜ 2006

Loengu ülevaade

- **Keerukus**: kui ruttu ülesannet lahendada saab?
  - Probleem
  - Põhi-ideed
  - Põhimõisted
  - Näited
- **Lahenduvus**: kas üldse ülesannet lahendada saab?
  - Probleem
  - Põhi-ideed
  - Põhimõisted
  - Näited

Some properties of an Algorithm

Kindlasti:

Deterministic: Given the same input, it produces the same output.

Finite: It can be described in a finite number of steps

Definite: Each step has a clearly defined meaning.

Soovitavalt:

Correct: It produces correct answers

Time Bounded: It eventually stops

Fast: it not only stops, but stops quickly

Some properties of an Algorithm

Kindlasti:

Deterministic: Given the same input, it produces the same output.

Finite: It can be described in a finite number of steps

Definite: Each step has a clearly defined meaning.

Verifitseerimine, testimine

Soovitavalt:

Correct: It produces correct answers

Lahenduvus

Time Bounded: It eventually stops

Fast: it not only stops, but stops quickly

Keerukus

Keerukus uurimisvaldkonnana

Keerukus on suur arvutiteaduse uurimisvaldkond.
Ingliskeelne harilik nimi: complexity theory

Tegeldakse keerukusega peamiselt kahes mõttes:

Kiirus ehk aeg (time): kui kiiresti algoritm peatub, kui kiireid algoritme on mingite ülesannete jaoks olemas?

Mälukasutus ehk ruum (space): kui palju mälu algoritm kasutab, kui väikese mälukasutusega algoritme on mingite ülesannete jaoks olemas?

Keerukus uurimisvaldkonnana

Vaadatakse eraldi:

Algoritmide keerukust (nii aja kui ruumi mõttes)

Ülesannete lahendamise keerukust:

kas mingi probleemide ehk ülesannete klassi jaoks on olemas algoritmi keerukusega vähem kui mingi f?

Raske juhus: kuidas näida, et mingi probleemide klassi K jaoks ei ole kiiremat algoritmi, kui kiirusega mingi F?

Algoritme on lõpmatult palju, neid ei saa kõiki läbi proovida

Järelikult tuleb kavalalt tõestada, et kiiremat algoritmi kui kiirusega F ei saa antud probleemi jaoks olla.

Hulgaliselt suuri, olulisi, lihtsalt formuleeritavaid probleeme on senini lahendamata (näiteks P=NP?)

Assessing time costs: what's important?

- Basic strategies to measure time costs:
    1. **use a clock** -- but faster machines invalidate comparisons
    2. **count individual steps** -- usually too precise

To evaluate **how effective an algorithm is** (not just particular code or a particular computer), **we don't worry about machine speed.**

But the time needed depends on the **size** of the problem. To evaluate how well an algorithm does *in general*, we consider what happens as the problem size increases.

Algoritmi kiirus antakse tüüpiliselt funktsioonina ülesande suurusest. **kiirus = F(n)**, kus n on ülesande suurus ja F on "lahendamise kiiruse funktsioon" ülesande jaoks suurusega n.

Keerukuse näide: the Fibonacci numbers

We introduce algorithms via a "toy" problem: computation of Fibonacci numbers. It's one you probably wouldn't need to actually solve, but simple enough that it's easy to understand and maybe surprising that there are many different solutions.

Leonardo of Pisa (aka Fibonacci) was interested in many things, including a subject we now know as population dynamics: For instance, how quickly would a population of rabbits expand under appropriate conditions?

As is typical in mathematics (and analysis of algorithms is a form of mathematics), we make the problem more abstract to get an idea of the general features without getting lost in detail:

We assume that a pair of rabbits has a pair of children every year.

These children are too young to have children of their own until two years later.

Rabbits never die.

The Fibonacci numbers

We then express **the number of pairs of rabbits as a function of time** (measured as a number of years since the start of the experiment):

F(1) = 1 -- we start with one pair
F(2) = 1 -- they're too young to have children the first year
F(3) = 2 -- in the second year, they have a pair of children
F(4) = 3 -- in the third year, they have another pair
F(5) = 5 -- we get the first set of grandchildren

In general F(n) = F(n-1) + F(n-2):

all the previous rabbits are still there (F(n-1))
plus we get one pair of children for every pair of rabbits we had two years ago (F(n-2)).

The algorithmic problem we'll look at today:
how to compute F(n)?

The Fibonacci numbers

The original formula seems to give us a natural example of recursion:

Algorithm 1:

```
int fib(int n)
{
   if (n <= 2) return 1
   else return fib(n-1) + fib(n-2)
}
```

The Fibonacci numbers

An example of the sort of basic question we study in this class is, **how much time would this algorithm take**?

How should we measure time?

The natural measure would be in seconds, but it would be nice to have an answer that didn't change every time Intel came out with a faster processor. We can measure time in terms of machine instructions; then dividing by a machine's speed (in instructions/second) would give the actual time we want.

However, it is hard to guess from a piece of pseudo-code the exact number of instructions that a particular compiler would generate. To get a rough approximation of this, we try measuring in terms of lines of code.

The Fibonacci numbers

Each call to fib returns either one or two lines.

If n <= 2, we only execute one line (the if/return).

if n = 3, execute 2 for fib(2), plus one each for fib(1) and fib(0): 4

It's like the rabbits! Except for the two lines in each call, the time for n is the sum of the times for two smaller recursive calls.

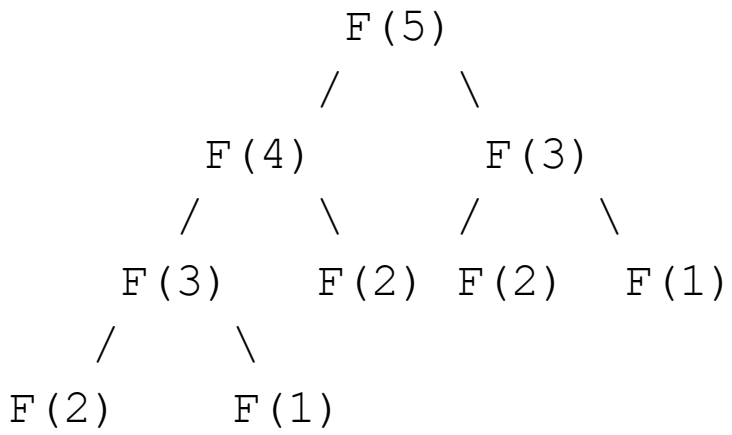
time(n) = 2 + time(n-1) + time(n-2)

In general, any recursive algorithm such as this one gives us a recurrence relation: the time for any routine is the time within the routine itself, plus the time for the recursive calls.

This gives in a very easy mechanical way an equation like the one above, which we can then solve to find a formula for the time. In this case, the recurrence relation is very similar to the definition of the Fibonacci numbers.

The Fibonacci numbers

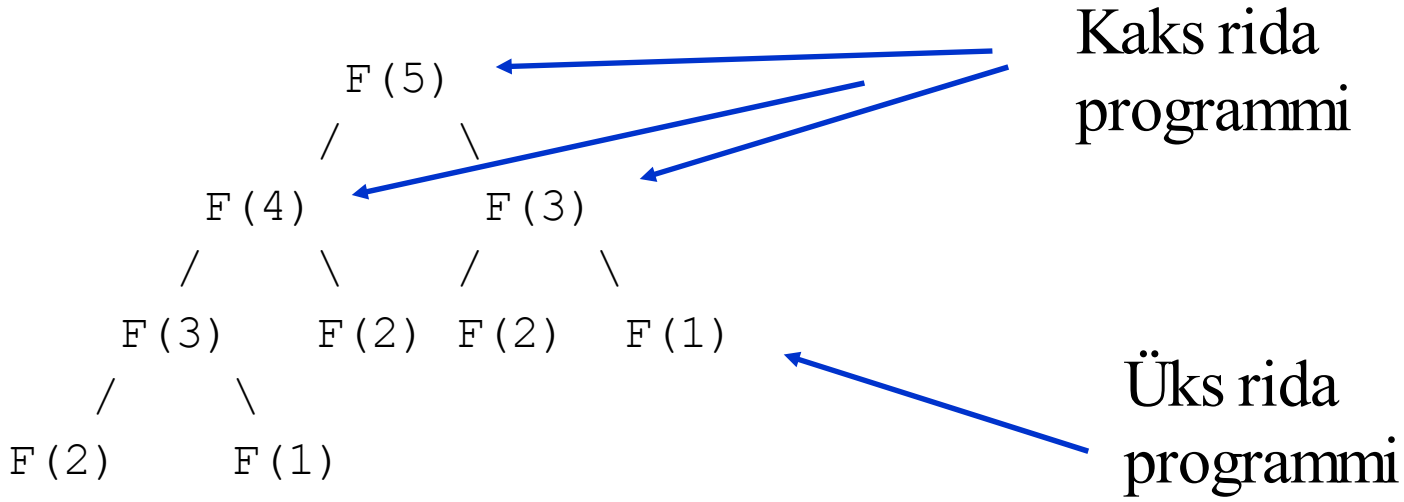
With some work, we can solve the equation, at least in terms of F(n): We think of the recursion as forming a tree. We draw one node, the root of the tree, for the first call then any time the routine calls itself, we draw another child in the tree.



The four internal nodes of this tree for fib(5) take two lines each, while the five leaves take one line, so the total number of lines executed in all the recursive calls is 13.

The Fibonacci numbers

With some work, we can solve the equation, at least in terms of F(n): We think of the recursion as forming a tree. We draw one node, the root of the tree, for the first call then any time the routine calls itself, we draw another child in the tree.



The four internal nodes of this tree for fib(5) take two lines each, while the five leaves take one line, so the total number of lines executed in all the recursive calls is 13.

The Fibonacci numbers

Note that, when we do this for any call to fib, the Fibonacci number F(i) at each internal node is just the number of leaves below that node, so the total number of leaves in the tree is just F(n).

So there are F(n) lines executed at the leaves, and 2F(n)-2 at the internal nodes, for a total of 3F(n)-2.

Let's double check this on a simple example:

time(5) = 3*F(5)- 2 = 3*5-2 = 13.

This is kind of slow **e.g. time(45) is over a billion steps**.

Maybe we can do faster?

The Fibonacci numbers: Dynamic programming

One idea: the reason we're slow is **we keep recomputing the same subproblems over and over again.**

Instead **let's solve each subproblem once and then look up the solution later** when we need it instead of repeatedly recomputing it.

Algorithm 2:

```
int fib(int n)
{
  int f[n+1];
  f[1] = f[2] = 1;
  for (int i = 3; i <= n; i++)
      f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

The Fibonacci numbers: Dynamic programming

This is an iterative algorithm (one that uses loops instead of recursion) so we analyze it a little differently than we would a recursive algorithm.

Basically, we just have to compute for each line, how many times that line is executed, by looking at which loops it's in and how many times each loop is executed.

Three lines are executed always.

The first line in the loop is executed n-1 times (except for n=1)

The second line in loop executed n-2 times (except for n=1) so:

time(n) = n-1 + n-2 + 3 = 2*n (except time(1)=4).

As an example for n=45 it takes 90 steps:

roughly 10 million times faster than the other program.

Running Times

(why people worry about algorithm complexity)

O

	n = 10	100	1,000	10,000	100,000	1,000,000
log n	3.3219	6.6438	9.9658	13.287	16.609	19.931
$\log^2 n$	10.361	44.140	99.317	176.54	275.85	397.24
sqrt n	3.162	10	31.622	100	316.22	1000
n	**10**	**100**	**1000**	**10000**	**100000**	**1000000**
n log n	33.219	664.38	9965.8	132877	$1.66*10^6$	$1.99*10^7$
$n^{1.5}$	31.6	10^3	$31.6*10^4$	10^6	$31.6*10^7$	10^9
n^2	100	10^4	10^6	10^8	10^{10}	10^{12}
n^3	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	1024	10^{30}	10^{301}	10^{3010}	10^{30103}	10^{301030}
n!	3 628 800	$9.3*10^{157}$	10^{2567}	10^{35659}	10^{456573}	$10^{5565710}$

Running times vs. problem size

Umbkaudsed astronoomilised piirid

Minimaalsed suurused:

Elektroni raadius ca 2.82 x $\mathbf{10^{-15}}$ m

Minimaalsed ajad:

Aeg, mis võtab valgusel elektroni raadiuse läbimine:

Valguse kiirus: 300.000.000 m/s = $\mathbf{3*10^{8}}$ **m / s**

Elektroni raadiuse läbimise aeg

(2.82 x $\mathbf{10^{-15}}$) / (3*$\mathbf{10^{8}}$) = ca = $\mathbf{10^{-15}}$ / $\mathbf{10^{8}}$ = ca = $\mathbf{10^{-23}}$ **sekundit**

Maksimaalsed mahud:

Elektronide ja prootonite koguarv universumis ca $\mathbf{10^{80}}$

Maksimaalsed ajad:

Universumi vanus ca 10^{10} aastat = ca = $\mathbf{10^{16}}$ **sekundit**

Universum kui hiidarvuti:

universumi eluea jooksul suudaks valgus läbida **ühe elektroni raadiust** kokku ca

$\mathbf{10^{16}}$ / $\mathbf{10^{-23}}$ = ca = $\mathbf{10^{39}}$ **korda**

universumi eluea jooksul suudaks valgus läbida **kõigi elektronide raadiusi** kokku ca

($\mathbf{10^{16}} * \mathbf{10^{80}}$) / $\mathbf{10^{-23}}$ = ca = $\mathbf{10^{119}}$ **korda**

Veel kiiremini kasvavad funktsioonid

Supereksponentsiaalsed funktsioonid (mitmekordne eksponent)

$$N^{N^{N}}$$

Näiteks: exp3(1) = 1, exp3(2) = 16, exp3(3) = $7*10^{12}$

Mittearitmeetilised funktsioonid (eksponentide kasvav torn):

$$\underbrace{N^{N^{\cdots^{N}}}}_{N^{N}\ \text{tk}}$$

Näiteks: expn(1) = 1, expn(2) = 16, expn(3) = 3 astmel $7*10^{12}$

Kombinatoorne plahvatus!!!

Hirmsad keerukusfunktsioonid praktikas

Eksponentsiaalse keerukusega ülesanded on praktikas väga levinud

Superekseponentsiaalse ja mittearitmeetilise keerukusega ülesanded tulevad samuti ette

Paljusid selliseid ülesandeid saab kavalate algoritmidega praktikas päris hästi lahendada ka üpris suure ülesannete (suure N) jaoks:

- lihtsalt sellised algoritmid ei tööta alati efektiivselt, vaid vahete-vahel.
- ka sellest on palju kasu, kui neid ülesandeid vahete-vahel õnnestub mõistliku ajaga lahendada
- Vahete-vahel kiirelt lahenduvate juhtude protsent võib praktikas osutuda üpris kõrgeks

Travelling salesman problem

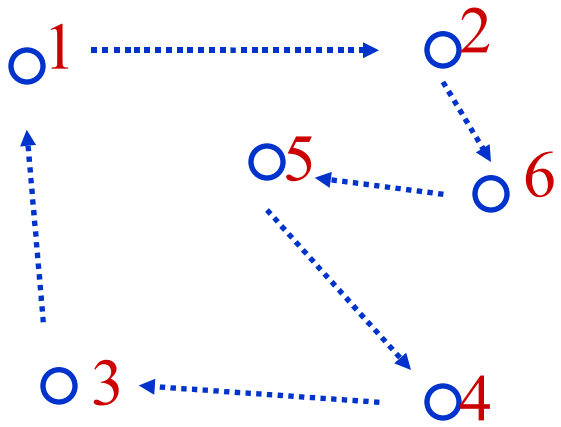
Want to visit some cities: New York, London, Tokyo, …

and come back to Singapore.

Each city visited exactly once (except first and last city is both Singapore)

Cost of travel between each pair of cities is given.

Aim: Find the cheapest route



Cost table:

	1	2	3	4	5	6
1	0	10	20	7	6	45
2		0	15	80	23	9
3			0	14	11	23
4				0	42	7
5					0	3
6						0

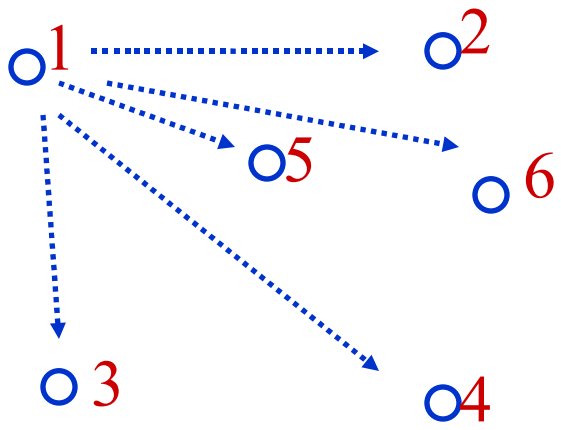
Is that a good route?

What is the cheapest route?

How many roads are there to try?

Example: six cities

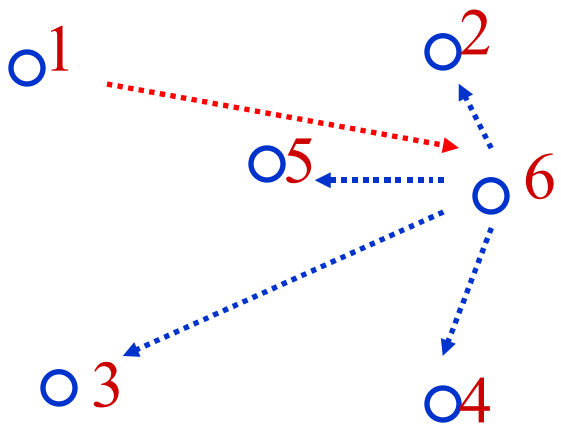
From the first city there are 5 choices



How many roads are there to try?

Let us pick one choice, say, to city 6.

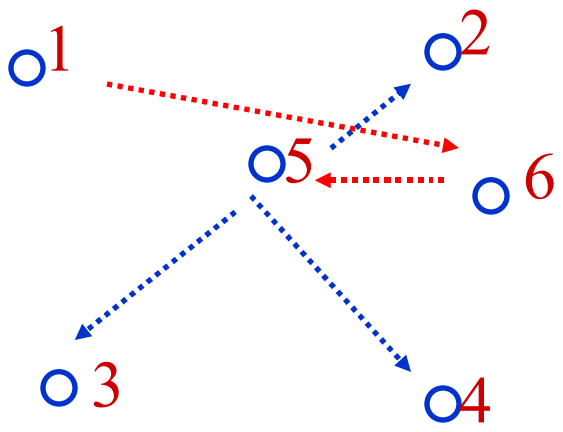
From city 6 there are 4 choices to proceed.



How many roads are there to try?

Let us again pick one choice, say, to city 5.

From city 5 there are 3 choices to proceed.



The amount of choices decreases with each new city.
Hence we have 5*4*3*2*1 different paths to measure, all in all.

Generally:
Nr of different paths is (N-1)!

Võimalike teede arv, mida läbi proovida

- Faktoriaal linnade arv miinus ühest:

Cities	Routes
1	1
2	1
3	2
4	6
5	24
6	120
7	720
8	5040
9	40320
10	362880
11	3628800

- A **10 city TSP** has 1,814,000 possible solutions
- A **20 city TSP** has 10,000,000,000,000,000 possible solutions
- A **50 City TSP** has

3 * 10 to power 64: ca 30,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000 possible solutions

Travelling salesman problem

Difficult to solve.

No polynomial time algorithm known.

Only exponential time algorithms known.

However, if given a tour, one can verify if the cost is less than X.

Why difficult: in case there are N cities to visit, there are **factorial(N)** possible routes: (if N=6, then: 5*4*3*2*1 routes to check)

Easy to verify answers. Difficult to find answers.

Easy ---> polynomial time

Difficult --> not polynomial time.

NP: Class of problems for which it is easy to verify the answers but exponential or factorial number of combinations to check

Travelling salesman problem is in NP.

Nobody has managed to prove (yet) that there are no polynomial algorithms for P.

Open question: does NP = P ?

Ebaintuitiivne kontranäide: maleprogramm

Näiliselt keeruline, keerukusteooria jaoks aga triviaalne ülesanne:

Leida mistahes maleseisus musta jaoks parim käik.

Miks see on keerukusteooria jaoks lihtne ülesanne?

Lahenduseks piisab hiigeltabelist, kus on igal real kirjas üks seis ja õige käik, mis sealt seisust teha.

Algoritm võtab seisu, vaatab tabelist vastava rea ja võtab seal antud käigu.

See võtab igakord samapalju aega (konstantne aeg) ja ei sõltu seisust.

Ei ole praktiline lahendus: tabel oleks nii suur, et me reaalselt ei suuda seda kuidagi täita ega arvuti mällu panna.

Teooria jaoks keeruline variant:

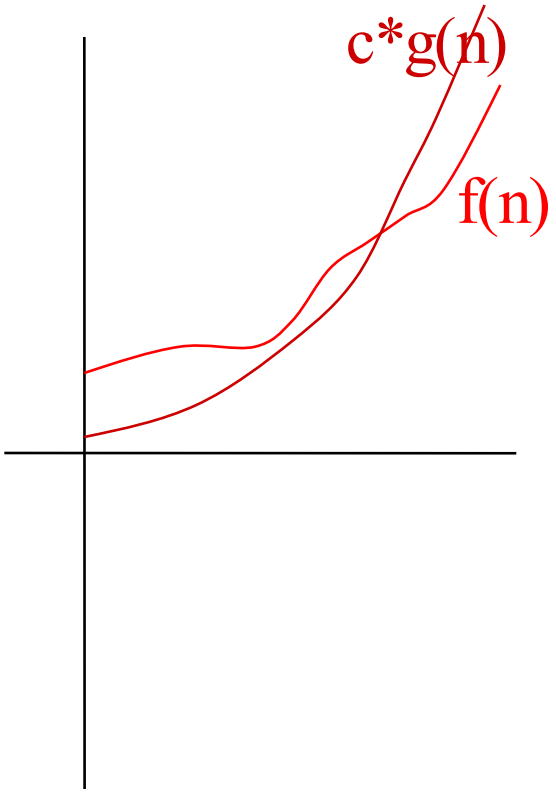
Võtame malemängu variandi, kus laua suurus ei ole piiratud. Saab mängida ka 10*10 laual, 100*100 laual jne.

Sel juhul ei ole enam teoreetiliselt võimalik teha (lõplikku) tabelit ja asi läheb keeruliseks (vähemalt eksponentsiaalselt keeruline)

Big-O Notation

- Way to express **upper bound** on a function
- **f(n) is O(g(n))** if
  f is bounded by a multiple of g(n) for large n
- **More formally:**
  there exist constants c > 0 and m > 0
  such that f(n) <= cg(n) whenever n >= m
- **Or using limit notation:**

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \quad 0 \text{ or a positive constant}$$



Analysing an Algorithm

- **Simple statement sequence**

  s_1; s_2; …. ; s_k

  - ***O(1)*** as long as *k* is constant

- **Simple loops**

  `for(i=0;i<n;i++) { s; }`

  where `s` is *O(1)*

  - Time complexity is $n * O(1)$ or ***O(n)***

- **Nested loops**

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++) { s; }
```

  - Complexity is $n * O(n)$ or $\boldsymbol{O(n^2)}$

Analysing an Algorithm

- **Loop index increases exponentially**

```
h = 1;
while ( h <= n ) {
  s;
  h = 2 * h;
}
```

  - `h` takes values 1, 2, 4, 8, 16, … until it exceeds *n*
  - There are $1 + \log_2 n$ iterations
  - Complexity ***O(log n)***

Analysing an Algorithm

Loop index depends on outer loop index

```
for(j=0;j<n;j++)
    for(k=0;k<j;k++){
      s;
    }
```

Inner loop executed

1, 2, 3, …., j times

Complexity ***O(n²)***

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Lahenduvus

- Teame, et iga probleemi jaoks ei leidu kiiret algoritmi.
- Kas aga iga probleemi jaoks on üldse olemas algoritmi, mis seda lahendab?
- Eeldame, et vaatame ainult probleeme, mis on täpselt ja üheselt kirjeldatud ja kus on lahendamiseks olemas piisavalt infot (a la travelling salesman, malemäng jne)
- **Selgub, et iga täpselt formuleeritud probleemi jaoks ei leidugi lahendavat algoritmi!**
- Vähe sellest: kui võtta “juhuslik” probleem, siis tõenäosus, et lahendav algoritm leidub, on lõpmatult väike!

Lahenduvus uurimisvaldkonnana

Algoritmi- ehk rekursiooniteooria on suur arvutiteaduse uurimisvaldkond.

Ingliskeelne harilik nimi: Algorithm theory, recursion theory

Lahenduvus: decidability või computability

Uuritakse, millistele ülesannetele on algoritme, millistele ei

Uuritakse, mis ülseande lahendamine taandub teisele ülesandele

Uuritakse lahendumis, poollahendumist, kreatiivseid hulki jne jne

Uuritakse lõpmatuse struktuuri, mis on kirjeldamatult keeruline

Uuritakse lahendumise struktuuri, mis on kirjeldamatult keeruline

Uuritakse loogikaklasside lahendumise taandumist muudele ülesannetele

....

Intuitiivne seletus lahendamatusele

- Saab näidata, et erinevaid probleeme on lõpmatult rohkem, kui erinevaid algoritme.
- Kuna probleeme on lõpmatult rohkem kui algoritme, siis iga probleemi jaoks lihtsalt “ei jätku” lahendavat algoritmi.
- Kuidas seda näidata (plaan):
  - Näitame, et **algoritme on sama palju, kui täisarve** (lihtne)
  - Näitame, et **probleeme on vähemalt sama palju, kui reaalarve** (veidi keerulisem)
  - Näitame, et **reaalarve on lõpmatult rohkem kui täisarve** (Cantori üks teoreeme)

Algoritme on sama palju (või vähem?) kui täisarve

- Iga algoritmi saab kirjutada mistahes programmeerimiskeeles.
- Valime näiteks C keele.
- **Iga C keelne programm on tekstifail, st üks pikk string.**
- String on näiteks “asas asss ddddd”.
- String koosneb järjestikustest baitidest, iga bait vahemikus 0-255
- **Iga string vastab ühele täisarvule:** vaatame stringi kui 256-süsteemis arvu, näiteks:
  - Baidid **0 22 64** annavad arvu **22*256+64**
  - Baidid **64 120 68** annavad arvu **64*256 ruudus + 120*256 + 64**
- NB! Iga string ei ole korrektne C programm. Vastupidi küll.

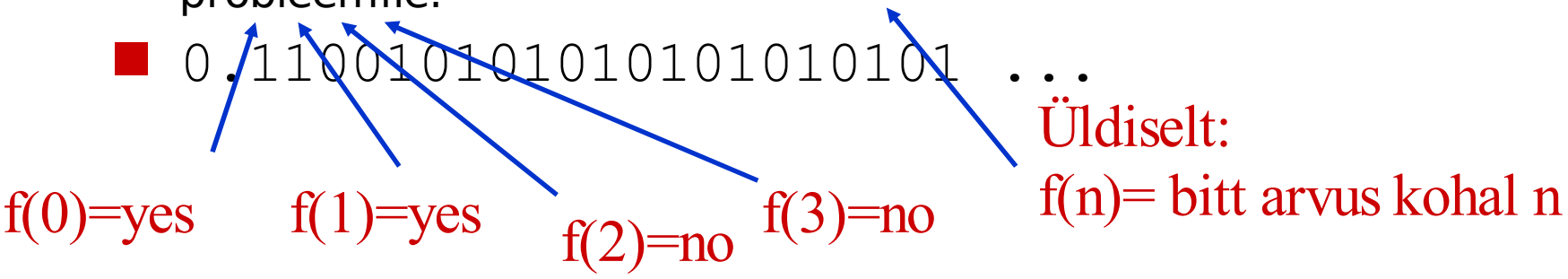
Probleeme on sama palju kui reaalarve

- **Mis on reaalarv? Arv, kus koma järel võib olla kuitahes palju komakohti.**
- **Näiteks**: 2,23242545344123123 1...
- 2, pi, ¾, ruutjuur kahest on kõik reaalarvud.
- Kahendsüsteemis reaalarvul on iga number kas 0 või 1, näiteks: 110.11010010110101111010101 01....

- Võtame ühe spetsiifilise klassi probleemidest: “yes-no” probleemid täisarvudel:

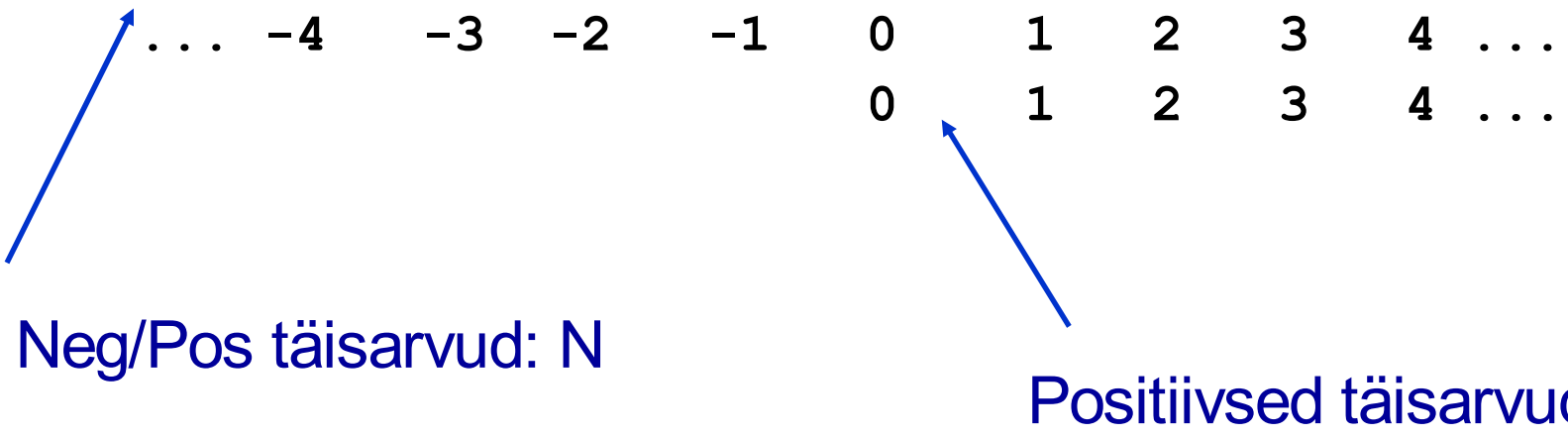
  Algoritm võtab sisendiks täisarvu ja peab vastama “yes” või “no”.

- Kui palju selliseid probleeme on?
- Iga kahendsüsteemis reaalarv alla ühe vastab ühele probleemile:
- 0.1100101010101010101 01 ...



Cantori teoreem: sissejuhatus

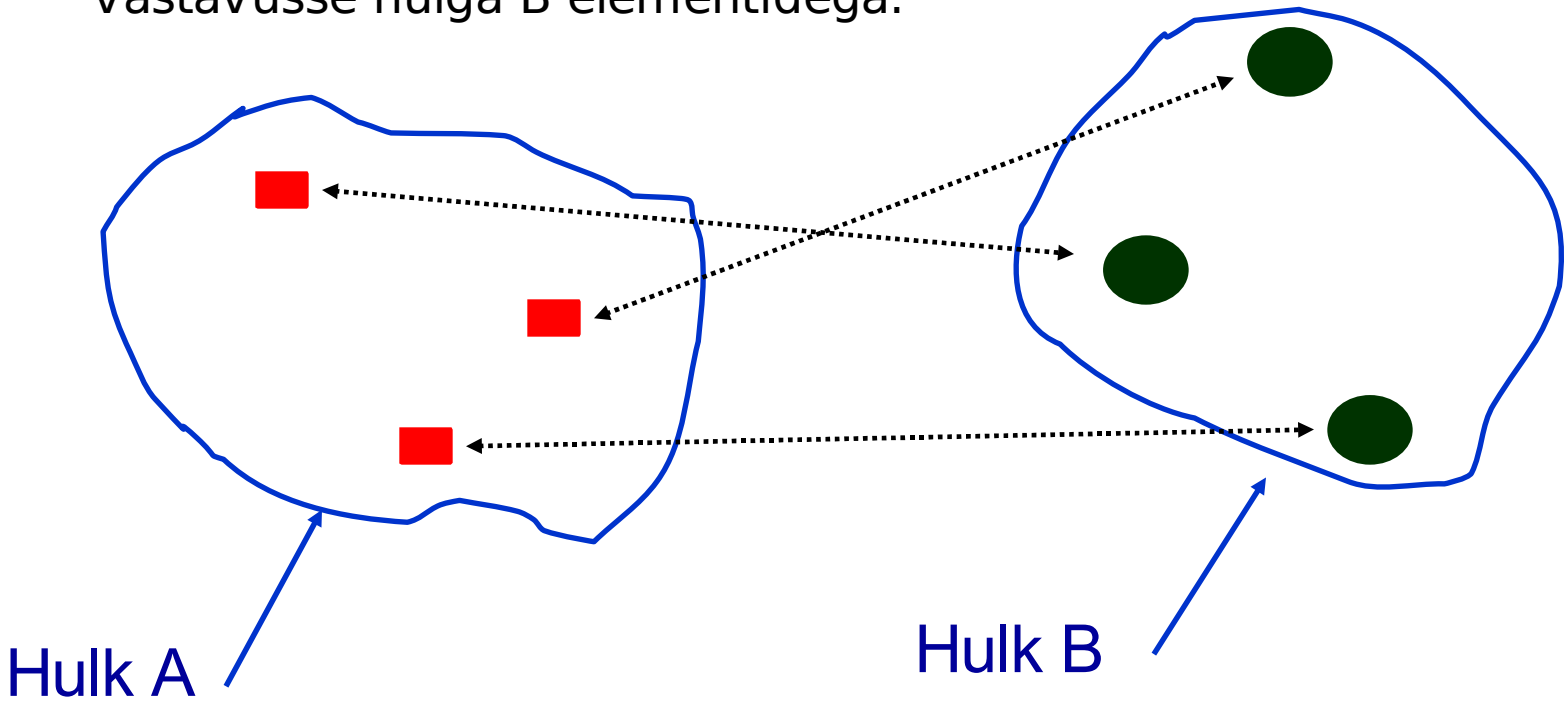
- **Reaalarvude hulk on suurem (võimsam) kui positiivsete täisarvude hulk.**
- Enne uurime, kuidas on lugu pos/neg täisarvudega ja murdudega.
- Pos/neg täisarve oleks justkui kaks korda rohkem, kui positiivseid täisarve??



Cantori teoreem: sissejuhatus

- Mida tähendab väide: “hulk A on sama suur (võimas) kui hulk B”?

- Seda, et hulga A kõik elemendid saab seda üksühesesse vastavusse hulga B elementidega:



Igale A elemendile vastab täpselt üks B element ja vastupidi

Cantori teoreem: sissejuhatus

- Kuidas seada üksühesesse vastavusse kahe lõpmatu hulga N ja Z elemendid?

N:	...	-4	-3	-2	-1	0	1	2	3	4 ...
Z:						0	1	2	3	4 ...

- N oleks justkui suurem kui Z?

Cantori teoreem: sissejuhatus

- Kuidas seada üksühesesse vastavusse kahe lõpmatu hulga N ja Z elemendid?

N:	...	-4	-3	-2	-1	0	1	2	3	4 ...
Z:						0	1	2	3	4 ...

- Seame vastavusse hoopis niimoodi:

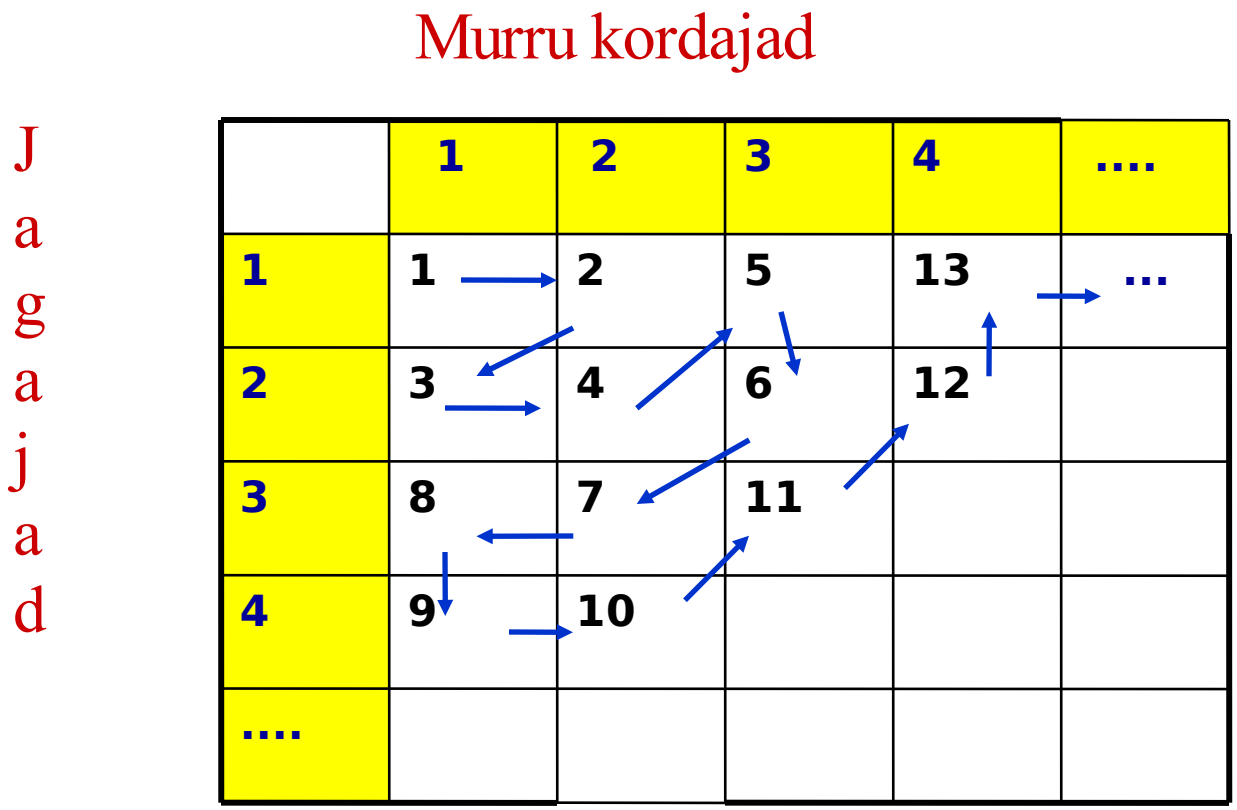
N:	0	1	-1	2	-2	3	-3	4	-4	...
Z:	0	1	2	3	4	5	6	7	8	...

- **Kuna N ja Z saab üksühesesse vastavusse seada, siis on nad sama suured (sama võimsad)!**

Cantori teoreem: sissejuhatus

- Murdarve oleks justkui lõpmatult rohkem, kui positiivseid täisarve?? Samuti vale!

Tegelikult on murdarvud vs pos täisarvud ükskühesed vastavuses:



Cantori teoreem: kõigepealt idee

- Koostame kõigi 1-st väiksemate reaalarvude tabeli:

Arvu kohad pärast koma

arvud						
	1	**2**	**1**	**5**	**6**	**...**
	3	**3**	**1**	**0**	**9**	**...**
	2	**8**	**3**	**5**	**6**	**...**
	3	**5**	**6**	**9**	**0**	**...**
					...	

- Ja nüüd konstrueerime diagonaali (**1 3 3 9 ...**) järgi uue arvu, liites diagonaali arvudele ühe (kui tulemus 10, võtame 0): saame **2 4 4 0 ...**

Cantori teoreem: detailselt läbi tehes 1

Arvu kohad pärast koma

arvud						
	1	2	1	5	6	…
	3	3	1	0	9	…
	2	8	3	5	6	…
	3	5	6	9	0	…
					…	

- **Konstrueerime uue arvu esimese numbri:**
  - Esimesel real tulbas 1 oli number 1.
  - Meie võtame 1+1 =2
  - Meie uue arvu esimene number on seega 2:
- **Meie uus arv: 0 . 2 …**

Cantori teoreem: detailselt läbi tehes 2

Arvu kohad pärast koma

arvud

1	**2**	**1**	**5**	**6**	**…**
3	**3**	**1**	**0**	**9**	**…**
2	**8**	**3**	**5**	**6**	**…**
3	**5**	**6**	**9**	**0**	**…**
				…	

- **Konstrueerime uue arvu teise numbri:**
  - Teisel real tulbas 2 oli number 3.
  - Meie võtame 3+1 =4
  - Meie uue arvu teine number on seega 4:
- **Meie uus arv: 0 . 2 4 …**

Cantori teoreem: detailselt läbi tehes 3

Arvu kohad pärast koma

arvud						
	1	**2**	**1**	**5**	**6**	**...**
	3	**3**	**1**	**0**	**9**	**...**
	2	**8**	**3**	**5**	**6**	**...**
	3	**5**	**6**	**9**	**0**	**...**
					...	

- **Konstrueerime uue arvu kolmanda numbri:**
  - Kolmandal real tulbas 3 oli number 3 .
  - Meie võtame 3+1 =4
  - Meie uue arvu kolmas number on seega 4:
- **Meie uus arv: 0 . 2 4 4 ...**

Cantori teoreem: detailselt läbi tehes 4

Arvu kohad pärast koma

arvud

1	**2**	**1**	**5**	**6**	**…**
3	**3**	**1**	**0**	**9**	**…**
2	**8**	**3**	**5**	**6**	**…**
3	**5**	**6**	**9**	**0**	**…**
				…	

- **Konstrueerime uue arvu neljanda numbri:**
  - Neljandal real tulbas 4 oli number 9 .
  - Meie võtame 9+1 = 10, meie võtame siis numbriks 0
  - Meie uue arvu kolmas number on seega 0:
- **Meie uus arv: 0 . 2 4 4 0 …**

Cantori teoreem: detailselt läbi tehes 5

Arvu kohad pärast koma

arvud						
	1	**2**	**1**	**5**	**6**	**...**
	3	**3**	**1**	**0**	**9**	**...**
	2	**8**	**3**	**5**	**6**	**...**
	3	**5**	**6**	**9**	**0**	**...**
					...	

- Meie uus arv oli: **0 . 2 4 4 0 ...**
- Kas meie uus arv on meie tabelis, st mõni rida tabelis?
- **Igatahes ei saa ta olla esimene rida:**
  - Esimesel real oli esimene arv **1**, meil aga on esimene arv **2**

Cantori teoreem: detailselt läbi tehes 6

Arvu kohad pärast koma

arvud

1	**2**	**1**	**5**	**6**	**…**
3	**3**	**1**	**0**	**9**	**…**
2	**8**	**3**	**5**	**6**	**…**
3	**5**	**6**	**9**	**0**	**…**
				…	

- Meie uus arv oli: **0 . 2 4 4 0 …**
- Kas meie uus arv on meie tabelis, st mõni rida tabelis?
- **Igatahes ei saa ta olla teine rida:**
  - Teisel real oli teine arv **3**, meil aga on teine arv **4**

Cantori teoreem: detailselt läbi tehes 7

Arvu kohad pärast koma

arvud						
	1	**2**	**1**	**5**	**6**	**…**
	3	**3**	**1**	**0**	**9**	**…**
	2	**8**	**3**	**5**	**6**	**…**
	3	**5**	**6**	**9**	**0**	**…**
					…	

- Meie uus arv oli: **0 . 2 4 4 0 …**
- Kas meie uus arv on meie tabelis, st mõni rida tabelis?
- **Igatahes ei saa ta olla kolmas rida:**
  - Kolmandal real oli kolmas arv **3**, meil aga on kolmas arv **4**

Cantori teoreem: detailselt läbi tehes 8

Arvu kohad pärast koma

arvud

1	**2**	**1**	**5**	**6**	**…**
3	**3**	**1**	**0**	**9**	**…**
2	**8**	**3**	**5**	**6**	**…**
3	**5**	**6**	**9**	**0**	**…**
				…	

- Meie uus arv oli: **0 . 2 4 4 0 …**
- Kas meie uus arv on meie tabelis, st mõni rida tabelis?
- **Igatahes ei saa ta olla neljas rida:**
  - Neljandal real oli neljas arv **9**, meil aga on neljas arv **0**

Cantori teoreem: detailselt läbi tehes 9

Arvu kohad pärast koma

arvud

1	**2**	**1**	**5**	**6**	**…**
3	**3**	**1**	**0**	**9**	**…**
2	**8**	**3**	**5**	**6**	**…**
3	**5**	**6**	**9**	**0**	**…**
				…	

- Selline asjade käik ei olnud juhus: me ise konstrueerisime oma arvu!
- Üldiselt on (meie enda konstruktsiooni-meetodi järgi) nii:
- **Igatahes ei saa ta olla N-s rida:**
  - N-ndal real oli N-s arv **X**, meil aga on N-s arv **X+1 (kui X= 9, siis 0)**

Cantori teoreem: detailselt läbi tehes 10

- **Kokkuvõttes:**

- Meie arv 0. 2 4 4 0 .... ei saa olla selles tabelis
- Kui meil oleks tabel kuidagi teisiti tehtud (arvud teises järjekorras) siis:
  - kui me jälle teeksime diagonaali järgi oma uue arvu, siis seda arvu

    ikka tabelis ei ole.
- Meie uus arv kahtlemata on reaalarv.
- Seega ei sisalda ükski 1-st väiksemate reaalarvude tabel kõiki reaalarve!
- Mis see siis tähendab:
  - Kõiki reaalarve ei saagi tabelisse panna
  - Iga tabeli N-s rida vastab täisarvule N
  - **Reaalarvude hulk on suurem (võimsam) kui täisarvude hulk**

Cantori teoreemi edasine jätk

Pane tähele, et:

- Iga kahendsüsteemis reaalarv 0 ja 1 vahel **vastab ühele täisarvude alamhulgale**: N-nda biti positsioon ütleb, kas arv N on seal hulgas või ei.

- **Näiteks:**
  - Paarisarvude hulgale vastab: `10101010101010101....`
  - Kõigi arvude hulgale vastab: `11111111111111111....`
  - Algarvude hulgale vastab: `01010101000101....`

- Cantori teoreem ütleb üldisemalt, et **mingi hulga H kõigi alamhulkade hulk on suurema võimsusega kui see hulk H.**

- **Tekivad lõpmatud ahelad üha suurema võimsusega hulkadest:**
- Täisarvude hulk N1 N1 alamhulkade hulk N2 N2 alamhulkade hulk N3 jne jne ....

Konkreetne näide mittelahenduvusest: Halting Problem

- Problem:You are an employee of the software company Gigasoft, which is writing a new operating system.
- **Your boss wants you to write a program that will take in a user's program and inputs and decide whether**
  - it will eventually stop, or
  - it will run infinitely in some infinite loop.
- If the program will run infinitely, your program will disallow the program to run and send the user a rude message.

- **Call this problem the Halting Problem.**

Attempts

- **Look for loops such as *while (statement)***
  - If variables in the statements are unchanged e.g.
    - *While (t<1)* with *t* is initialized to *0* but never used in the loop
    - No statements to get out of the loop, such as *goto* or *break*
  - The program will not halt.
- **What about programs like:**

```
TestGolbach
    i=2; c=true;
    while (c==true)
        i=i+2;
        find all primes smaller than i, put in a set S;
        is_sum = false;
        for each pair (p,q) of primes in S
            if (i==p+q) is_sum=true;
        if (is_sum==false) c=false;
```

  - Will TestGolbach halt?

Is there a solution?

- **If you can solve the halting problem, you can solve Golbach's conjecture:**
  - Every even number greater than 2 can be represented as a sum of two primes.
- **Golbach's conjecture is an unsolved problem in mathematics**
- **What does this tell you about the halting problem?**
- **What should you do?**
  - One possibility: try to show that the halting problem is not solvable.

Proof: start

- Assume that it is possible to write a program to solve the Halting Problem.
- Denote this program by **HaltAnswerer(*prog*,*inputs*).**
- **HaltAnswerer(*prog*,*inputs*) will**
  - **return *yes* if *prog* will halt on *inputs* and**
  - ***no* otherwise.**
- A program is just a string of characters
  - E.g. your Java program is just a long string of characters
- An input can also be considered as just a string of characters
- So HaltAnswerer is effectively just working on two strings

Proof part 2:

- We can now write another program **Nasty(prog)** that uses **HaltAnswerer** as a subroutine
- The program **Nasty(prog)** does the following:

  [1] If HaltAnswerer(prog,prog) returns yes,
  Nasty will go into an infinite loop

  [2] If HaltAnswerer(prog,prog) returns no,
  Nasty will halt

- Consider what happens when we run **Nasty(Nasty)**.
- If **Nasty loops infinitely**,
  - HaltAnswerer(Nasty,Nasty) returns no which by [2] above means Nasty will halt.
- If **Nasty halts**,
  - HaltAnswerer(Nasty,Nasty) will return yes which by [1] above means Nasty will loop infinitely.

- Conclusion: Our assumption that it is possible to write a program to solve the Halting problem has resulted in a contradiction.

Diagonalization for halting problem: try it out!!

- **Each program can be represented by a string and each string can be represented by a natural number**
- **Create a table for all programs with a single input where the entry *(i,j)* is**
  - *H* if program *i* halts when program *j* is used as input
  - *NH* if program *i* does not halt when program *j* is used as input
- **Where is *Nasty* in this list?**

Kuulus näide matemaatikast

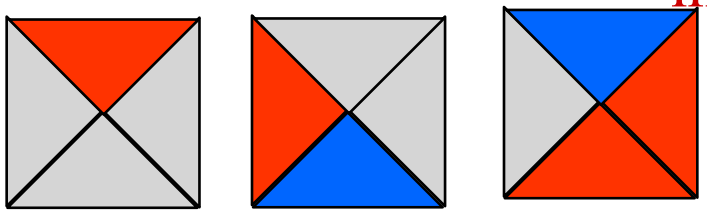
- In 1900, mathematician David Hilbert identified 23 mathematical problems and posed them as a challenge for the coming century.
- **The tenth problem asks for an algorithm to test whether a polynomial has an integral root**
- Apparently, Hilbert assumed that such an algorithm must exist.
- **This problem is unsolvable** (Matijasevic 1970)

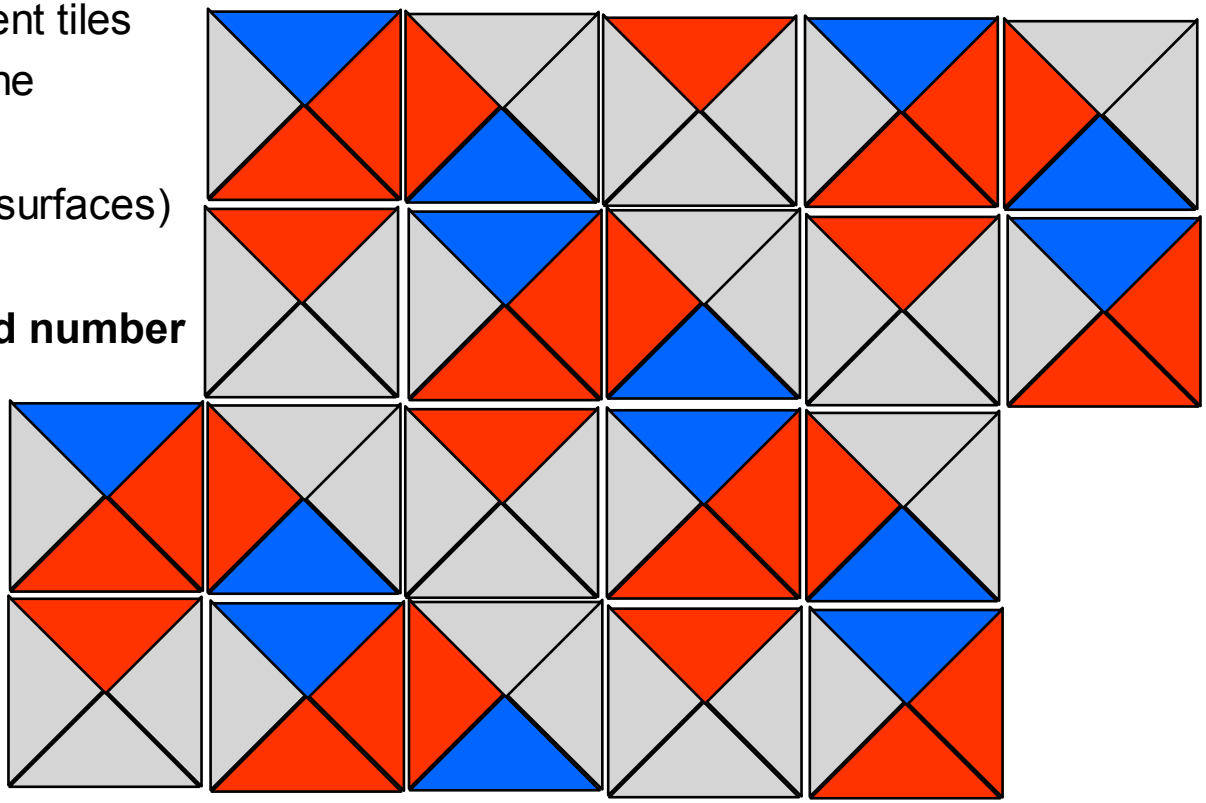
Lihtne näide geomeetriast: “tiling problem”

Assume that you have
a finite set of tiles
(which cannot be rotated)
and you would like
to know if you can tile
an area using those
tiles (adjacent tiles
must have the
same color
on adjoining surfaces)

An **unlimited number of tiles** is available of each kind

Three tiles given in this example

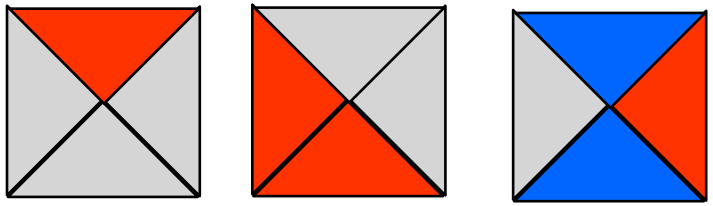


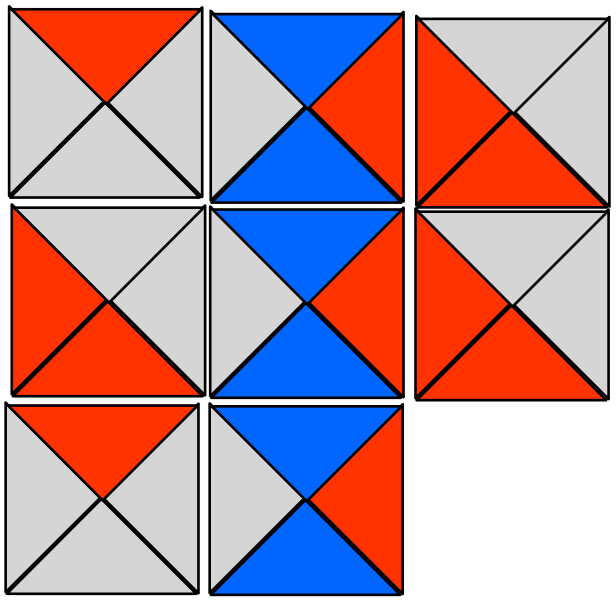


Tiling problem: lõpliku pinna katmine

- **Can I use this set of tiles to tile the floor of a house?**
  - **Lahenduv:** proovime järele kõikvõimalikud paigutused
  - Kuna maja põrand on lõplik, siis proovitavaid variante on lõplik (kuigi väga suur) hulk.
  - Võib mh meelde jätta, et see on NP-täieliku keerukusega ülesanne.
- **Selgub, et mõne plaatide hulgaga saab, mõnega ei!**
  - Lihtne on näiteks juhtum, kus on ainult ühte tüüpi plaate: valge.
  - Aga vaatame järgmist näidet (kolm erinevat plaaditüüpi) , kus ruudukujulise pinna plaatimine ei õnnestu.

Another example tile set:





Error

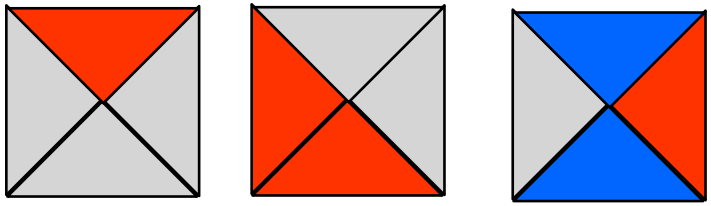
No Tiling for even small areas

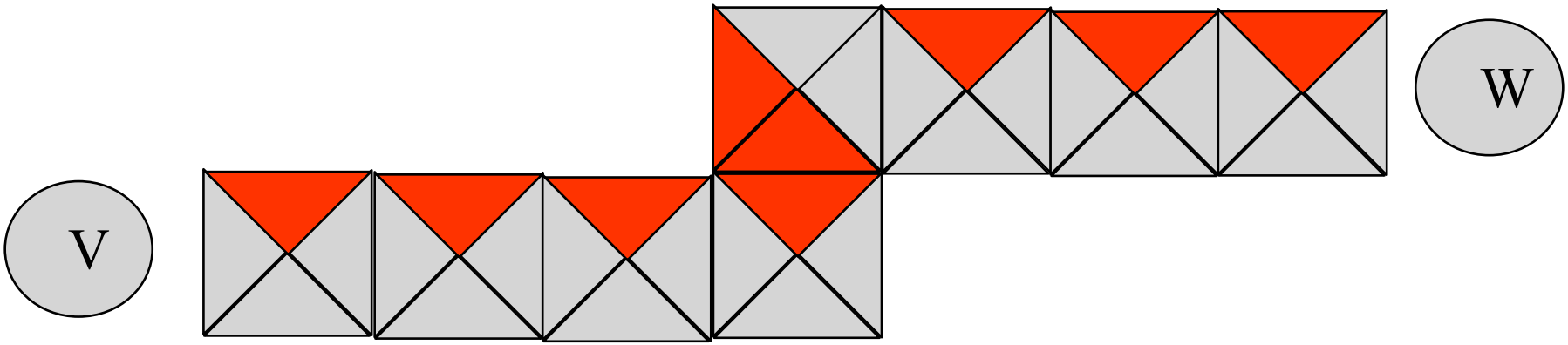
Tiling problem: undecidable question

- **Can a room of any size be tiled using this set of tiles?**
- **St, pind suurusega 2*2, pind 3*3, pind 4*4, jne jne ....**
  - **Ei ole lahenduv!**
  - Pane tähele, et **järeleproovimise meetod** enam ei anna kindlaid tulemusi, sest järgi tuleks proovida lõpmatult palju pinnasuurusi.
  - Pane tähele ka seda, et kui vastus on negatiivne (mingi suurusega N*N pinda ei saa meie plaatidega katta), siis mehaaniline järeleproovimine lõpuks selle avastab.
  - Raskus tekib, kui tegelikult saab kõiki pindu katta: järeleproovimise meetod proovib üha suuremaid ja suuremaid pindu, lõpmatuseni.

Analoogiline näide: “Domino Snakes”

- **Is it possible to connect V to W using a tile snake?**





“Domino Snakes” kohta saab tõestada, et:

Lets look at three cases:

- If the plane to lay down the snake in is finite?
  - **trivially decidable**
- If snakes can go anywhere in the plane?
  - **decidable**
- If snakes can only go in half of the plane?
  - **undecidable**

Kuulus näide matemaatikast

- In 1900, mathematician David Hilbert identified 23 mathematical problems and posed them as a challenge for the coming century.
- **The tenth problem asks for an algorithm to test whether a polynomial has an integral root**
- Apparently, Hilbert assumed that such an algorithm must exist.
- **This problem is unsolvable** (Matijasevic 1970)

Poollahenduvus

- Olgu ülesandeks tuvastada, **kas täisarv X kuulub mingisse lõpmatusse täisarvude alamhulka H**.
  - Mõne H jaoks on ülesanne **lahenduv**: näiteks, kui H on paarisarvude hulk, kui H on algarvude hulk jne,
  - Mõne H jaoks ülesanne **ei ole lahenduv**: näiteks, kui H on arvude hulk, millele vastavad programmid peatuvad.

- **Poollahenduvus** tähendab, et kui X juhuslikult kuulub hulka H, siis me saame seda algoritmiga alati näidata. Kui ei kuulu H-i, siis ei saa alati.

- Peatumisprobleemi puhul: paneme X-le vastava programmi käima ja kui ta peatub, siis loomulikult teame, et ta kuulub hulka H
  - Kui ta aga ei peatu, siis meil ei ole kindlat viisi aru saada, et ta ei kuulu hulka H.
  - **Peatumisprobleem on poollahenduv**.

- **On olemas ülesandeid, mis ei ole ka mitte poollahenduvad.**

Lahenduvus: muud

- **Vanad “vist ekslikud” oletused:**

2. Mathematics is consistent. Roughly this means that we cannot prove a statement and its opposite; we cannot prove something horrible like 1=2.
3. Mathematics is complete. Roughly this means that every true mathematical assertion can be proven i.e. every mathematical assertion can either be proven or disproven.
4. Mathematics is decidable. This means that for every type of mathematical problem there is an algorithm that, in theory at least, can be mechanically followed to give a solution. We say “in theory” because following the algorithm might take a million years and still be finite

Lahenduvus: muud

- **Hiljem selgus, et:**

- **In 1930, Kurt Godel shocked the world by proving that 1 and 2 cannot both be true**

  Either you can prove false statements or there are true statements that are not provable.

  Most people believe that mathematics is incomplete rather than mathematics is inconsistent

- **Turing was one of the people who showed that (3) is false by his work on Turing machines and the halting problem**